Programming the Data Structure Accelerator

Richard Zippel[*]
Department of Computer Science
Cornell University, Ithaca, NY 14863 USA
rz@cs.cornell.edu

(to the 6th pr[illegible] / 1990 / (Single instruction multiple data (SIMD) Cr

Abstract

We present a fine grained, massively parallel SIMD architecture, called the *data structure accelerator*, and demonstrate its use in a number of problems in computational geometry. This architecture is extremely dense and highly scalable. Systems of 10^6 processing elements can be feasibly embedded in workstations. We advocate that this architecture be used in tandem with conventional, single sequence machines and with small scale, shared memory multiprocessors. A language for programming such heterogeneous systems is presented that smoothly encorporates the SIMD instructions of the data structure accelerator with conventional single sequence code.

1 Introduction

There has been a significant body of work on single instruction, multiple data (SIMD) computer architectures in the past. This work ranges from the MPP machine developed at Goodyear [4] to the Connection Machine[3] and the Masspar MP-1 today . These machines are generally viewed (and sometimes even advertised) as large "supercomputers." They are intended to be used for large problems that are not practical on smaller machines. However, these SIMD machines are not uniformly better than conventional processing elements or the new generation of MIMD processors for all problems. It is rarely cost effective to couple the large SIMD machines with other types of processing elements, so that a combined, heterogeneous machine can be applied to a problem—each component performing those computations at which it is best. Furthermore, existing SIMD machines have a relatively modest number of processing elements (the Connection Machine can handle up about $10^{4.8}$).

The chunks of computation performed on the current generation SIMD machines tend to be quite large. There is relatively little cooperation between the SIMD machine and the host machines during the computation. This reinforces the division between SIMD computations and more conventional approaches.

We feel this division is counterproductive. To illustrate this a few algorithms are presented in Section 4 that use a mixture of SIMD and SISD constructs to achieve high performance. Interestingly, these algorithms are quite simple compared to the optimal, single sequence algorithms for the same problem, and yet the SIMD algorithms perform substantially better. The problems these techniques solve arise as parts of much larger problems, such as mechanical simulation, that are quite difficult to parallelize using SIMD techniques. It would be wildly impractical to devote a Connection Machine their resolution in most cases.

In this paper we suggest that the real role for SIMD architectures is not as "stand-alone supercomputers," but as integral components of heterogeneous machines that consist of both SIMD and SISD (or MIMD) components—each component responsible for the portions of a computation at which they are most effective. This often means managing large, memory resident data structures, or performing simple operations on large blocks of data. Thus, a natural way to merge a SIMD architecture with a conventional one is to integrate the SIMD processing elements into the memory system.

We argue for simple SIMD architectures that can be built relatively cheaply and with very high density. We are interested developing systems with upwards of 10^6 processing elements that can be used in personal workstations and upwards of 10^8 for "supercomputing applications." The individual elements of such SIMD architectures must be quite simple to have this type of density and their interconnect must also be simple to allow for scalability and the size system we are interested. We have developed a class of SIMD architectures that meets these criteria which we call *Data Structure*

[*]This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, the National Science Foundation through grant DMC-86-17355 and the Office of Naval Research through contract N00014-89-J-1946.

Accelerators (DSA).[1] In Section 2 we present their organization in detail.

One of the problems with dealing with heterogeneous architectures is the difficulty of expressing algorithms clearly and succinctly. In Section 3 we describe a few natural extensions of a hypothetical C-like langauge that simplifies the description of data parallel computations. This extended language is unique in that it allows one to express cooperative algorithms that are executed on a heterogeneous computer consisting of both a SIMD component and conventional single sequence component.

Having SIMD computation cheaply available affects the type of algorithms that are used. Many of the complex algorithms developed for searching and managing data structures are no longer necessary because the data structures can be managed by the DSA and all elements handled in parallel. This is demonstrated in Section 4 where we discuss a few problems in computation geometry.

2 The Data Structure Accelerator Architecture

The Data Structure Accelerator (DSA) is a class of SIMD architectures that is extremely dense, easily scalable and that has been optimized to efficiently perform functions that are difficult for conventional processing systems. The DSA's processing elements (PE's) are sufficiently compact that machines with upwards of 10^7 processing elements are feasible—well into the massively parallel regime.

The processing elements are connected in a low dimension, rectangular grid. This type of interconnect is much cheaper, and can be scaled upwards much better than the boolean n-cube network used by machines like the Connection Machine. Although using this simple interconnect scheme makes a few problems impractical, we felt that the improved scale and cost of the resulting system more than compensated. For many applications a one dimensional interconnect is sufficient. For some problems in vision and computational geometry two dimensional interconnects are useful. In principle, higher dimensional interconnects could be used, but their impact on pin count and PE/chip density is severe. For now we are only considering one and two dimensional data structure accelerators.

To keep the processing elements small, we have decided to restrict them to single bit width. This minimizes the number of operations that need to be incorporated in each PE, but they are sufficiently powerful for most applications. Because the DSA is often used to manage large tables, we have included content addressable memory in the DSA architecture. An example of such an element used in a one dimensional linear array is shown in Figure 1. Each processing element is called a *line*. A line contains some amount of content addressable memory (CAM) and random access memory (RAM). A particular data structure accelerator is characterized by four parameters: the number of lines in the DSA (ℓ), the dimensionality of their interconnect, the number of bits of CAM (m) and the number of bits of RAM (n). The *SelectWord*, which controls which PE's participate in an operation, is shared by all the lines of the DSA.

Generally, the dimensionality of the DSA is understood (it is usually one) and the number of lines is not important to the algorithms. However, the size of the CAM and RAM structures is important. Thus, we say that a data structure accelerator has *parameters* (m, n) when each line has m bits of CAM and n bits of RAM. The amount of CAM and RAM associated with each line of a DSA can be optimized for different applications and can range from DSA systems with all CAM and no RAM to all RAM and no CAM. The Smart Memories Project at MIT has built a 64 line (32, 4) DSA chip [6, 8], a 256 line (32, 4) DSA chip [1] and a board that implements a 4096 line (32, 4) DSA. With current technology, devices with thousands of lines are not impractical and systems in the range of 10^6 to 10^8 PE's are feasible.

The data structure accelerator uses a three valued logic for two purposes: to indicate which elements of an array execute particular instructions and within the CAM cells, to increase their flexibility. One unit of this three valued system is called a *trit*. Each trit can assume one of the values `0`, `1` or `X`. The only binary operation we perform with trits is *equivalence*, which obeys the following "truth" table.

≡	0	1	X
0	1	0	1
1	0	1	1
X	1	1	1

The DSA architecture is capable of performing five basic instructions: **`select`**, **`write`**, **`match`**, **`operate`** and **`readout`**. The **`select`** instruction specifies which processing elements participate in the next sequence of instructions by writing its operand, which is a trit string, into the *SelectWord* of the DSA. Until the next **`select`** instruction, only lines whose address matches

[1]Earlier versions of this work at MIT referred to this architecture as a Database Accelerator. Since this effort is directed towards "in memory" databases our original choice of names was somewhat misleading. To correct this we have chosen the name *data structure accelerator* which is more suggestive.

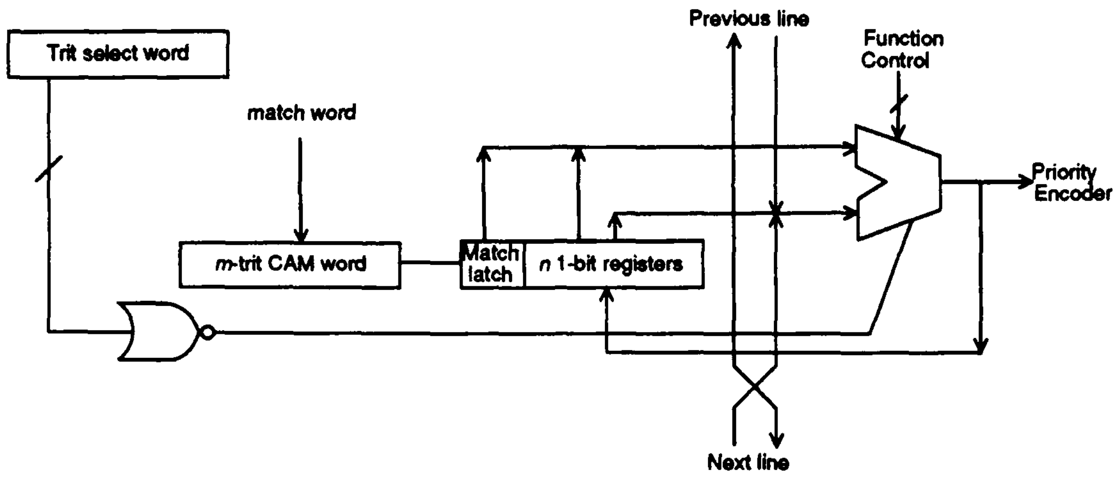


Figure 1: MIT Database Accelerator Architecture

the contents of the select word perform any DSA operation. Thus to have all lines active, the select word is filled with X's. To have the even lines participate, a XXXX···X0 is used, and if 3, 11, 19 and 27 are to participate, the select word will contain 0···0XX011.

The other four instructions are executed in parallel by each of the selected processing elements in a SIMD fashion. The **write** instruction writes its operand into the CAM word of the selected processing element(s). Since the *SelectWord* can contain X's a **write** instruction can cause the CAM of more than one PE to be modified.

The **match** instruction includes a data word that is matched against the contents of the CAM words of each of the selected PE's using the equivalence function given above. The result of the match is then written into the *MatchLatch* where it can use used by the **operate** instruction. No data is transmitted out of the DSA by a **match** instruction.

The **operate** instruction causes each selected PE to perform a boolean operation on the contents of two registers and store the result in a third. This is a three operand instruction. As shown in Figure 1, one of the operands can come from the *MatchLatch* or a register of an adjacent PE. The result of the boolean operation is also latched by the priority encoder.

The contents of the priority encoder are read using the **readout** instruction. The **readout** instruction returns the address of one of the lines whose priority encoder latch is set. At the same time it clears that particular priority encoder latch. Consequently, successive **readout** instructions return the addresses of the lines whose priority encoders contain a one. A special code is returned if all priority encoder latches contain zeroes.

Even though a particular set of parameters must be chosen when building a DSA system, the user can simulate a DSA with a different set of parameters at surprisingly small cost as shown in following table. Each entry indicates the cost of simulate the particular operation on a DSA with parameters $(km, \ell n)$. This ability to emulate DSA's with different parameters is one of the most important differences between the DSA and previous CAM designs.

Operation	$(km, \ell n)$	(m, n)	$(0, \ell n)$
match	1	$2k-1$	$2km-1$
operate	1	ℓ	ℓ
write	1	j	km

3 An Algebraic Language for Specifying DSA Operations

We do not believe the DSA should be viewed as a universal computing engine but rather as a component of a heterogeneous computing system, where the DSA is used as a slave of some host processor or perhaps shared for among several processors. The host, or its delegate, is responsible for sequencing the DSA instructions and performing those data operations for which a SISD or MIMD machine is preferable.

Thus algorithms that utilize the DSA are a mixture of DSA instructions and conventional single sequence processor instructions. Rather than expressing these algorithms in a mixture of low level DSA instructions and some high level language for the single sequence portion of the instruction stream, we have developed a set of high level extensions to an algebraic programming language in which all the DSA operations can be used effectively. This approach allows us to intertwine DSA operations with more conventional programming mechanisms including multiprocessing extensions. This section does not give a complete description of the language, which is still evolving, but rather describes the major features and provides enough information to make the examples in the later section clear.

The components of the DSA description language have five basic components.

- Declarations that describe the allocation of DSA lines to different DSA arrays, and the allocation of CAM and RAM of the lines of a DSA array to various tasks.
- Basic operations for comparing the CAM contents with fixed data and performing boolean and arithmetic operations with the contents of the RAM.
- Loop abstractions that cause operations to be performed on blocks of DSA lines.
- A mechanism for describing algorithms best expressed as state transition tables.
- A library of higher level functions.

Each of these components is described in one of the following subsections. It is important to notice that our language intersperses DSA operations and conventional SISD operations. We believe this allows our description language to be more expressive, and properly leaves to the compiler the problems of the separating the operations that are performed on the DSA form those performed on the host processor.

3.1 Declarations

The N processing elements in a DSA are identified by their coordinates within their interconnection grid. These coordinates are used as a subscript. For one dimensional DSA's this is just an integer from 0 to $N-1$. Higher dimensional arrays use vector subscripts.

For instance, the RAM of the $\vec{i}^{th}$ line is written as $R_{\vec{i}}$, while the CAM of each line is written as $K_{\vec{i}}$. In the case of a one dimensional DSA we will denote the RAM by R_i and the CAM by K_i. In the two dimensional case we use $R_{i,j}$ and $K_{i,j}$. The individual bits of the RAM can be referenced by $R_{\vec{i}}[0], \ldots, R_{\vec{i}}[n]$. The R_i, K_i and $\mathcal{M}_i$ registers are not normally available to the programmer. Instead, the programmer uses variable declarations to indicate the resource requirements of his or her algorthm, and the compiler allocates them from the available resources. This insulates the programmer from the complications of emulating resources with what is actually available.

Collections of DSA lines are called *DSA arrays*. Individual DSA arrays are allocated as if they were arrays, but whose elements are declared using **DSAstruct**, which indicates to the compiler the RAM and CAM requirements of each line. For instance,

```
DSAstruct interval {
    CAM color[5];
    RAM selected, min[16], max[16];
    States S ∈ {inside, outside, unknown};
    }
```

defines the structure of a line of a DSA array. Only one CAM variable is allocated, **color**, which is 5 bits long. Three RAM variables are allocated, two of 16 bits and one of 1 bit. The **States** declaration indicates the allowable states of each line when programming the DSA using state transition techniques. The state transition techniques and the **States** declaration are described fully in Section 3.4. A DSA line with this structure will have at least 5 bits of CAM and 35 bits of RAM.

DSA arrays are collections of one or more *primitive blocks*, where each primitive block is a set of 2^k DSA lines on a 2^k boundary. Blocks of DSA lines are only allocated in sizes that are a power of 2 because of the organization of the decoders. Odd sized DSA array's are allocated as sets of primitive blocks. Primitive blocks are identified by the selector word that spans their elements. Thus $\mathtt{100XXX}_2$ represents the primitive block that extends from lines 32 through 39, inclusive. To indicate that the index i lies within this primitive block, we write $i \in \mathtt{100XXX}_2$.

A DSA array with 100_{10} elements would consist of primitive blocks of size 64, 32 and 4. It would be represented by the union of the identifiers for its constituent primitive blocks. For instance, for the DSA array defined by the statement:

```
DSAstruct interval Table[100];
```

we would have

$$\mathtt{Table} = \mathtt{100XXXXXX}_2 \cup \mathtt{1010XXXXX}_2 \cup \mathtt{1110000XX}_2$$

The lines of **Table** each contain at least 5 bits of CAM and 35 bits of RAM. Subscripts are used to identify lines, so the fifth line of **Table**, all 40 bits of it are referred to as $\mathtt{Table}_5$. Particular variables are referred to concatenating the DSA array name with the variable name, separated by a slot, *e.g.* **Table.min** or $\mathtt{Table.min}_i$.

Set intersection and complement can also be used to describe DSA arrays. For instance, the even lines of **Table** might be denoted by

$$\begin{aligned}\texttt{Table} \cap \texttt{XXXXXXXX0}_2 &= (\texttt{100XXXXXX}_2 \cup \texttt{1010XXXXX}_2 \cup \texttt{1110000XX}_2) \\ &\quad \cap \texttt{XXXXXXXX0}_2 \\ &= \texttt{100XXXXX0}_2 \cup \texttt{1010XXXX0}_2 \cup \texttt{1110000X0}_2\end{aligned}$$

and the lines whose indices are not multiples of 4 by

$$\begin{aligned}\texttt{Table} \cap \overline{\texttt{XXXXXXX00}_2} &= (\texttt{100XXXXXX}_2 \cup \texttt{1010XXXXX}_2 \cup \texttt{1110000XX}_2) \\ &\quad \cap (\texttt{XXXXXXX1X}_2 \cup \texttt{XXXXXXX01}_2) \\ &= \texttt{100XXXX0X}_2 \cup \texttt{100XXXX01}_2 \cup \texttt{1010XXX1X}_2 \\ &\quad \cup \texttt{1010XXX01}_2 \cup \texttt{11100001X}_2 \cup \texttt{111000001}_2\end{aligned}$$

Each of these three operations can be performed formally on the selector bit strings as follows. We consider the union of two selector bit strings $R = r_1 \cdots r_k$ and $S = s_1 \cdots s_k$. Assume R and S differ in just one bit position, so $r_i = s_i$ for $i \neq \ell$. There are then three possibilities:

$$R \cup S = \begin{cases} R & \text{if } r_\ell = \texttt{X} \\ S & \text{if } s_\ell = \texttt{X} \\ r_1 \cdots r_{\ell-1}\texttt{X}r_{\ell+1} \cdots r_k & \text{otherwise} \end{cases}$$

The intersection of R and S can be performed on bit by bit basis. Assume r_i and s_i differ. If neither is an X then the intersection of R and S is the empty set. Otherwise, the intersection uses the bit which is not equal to X.

The complement of R is a union of ℓ bit strings, where ℓ is the number of 0's and 1's in R. To see this examine the simple case $R = \overline{\texttt{X000}_2}$. We can trivially write R as a union of 7 bit strings, where each has one X in the same position. These strings are listed in on the left hand side of the double bars in the table below. On the right hand side, are given the three simplified bit strings whose union is R.

$\texttt{X100}_2$	$\texttt{X101}_2$	$\texttt{X110}_2$	$\texttt{X111}_2$	$\texttt{X1XX}_2$
$\texttt{X010}_2$	$\texttt{X011}_2$			$\texttt{X01X}_2$
$\texttt{X001}_2$				$\texttt{X001}_2$

These rules allow us to reduce all combinations of unions, intersections and complements of selector bit strings to unions of bit strings, *i.e.*, conjunctive normal form.

3.2 Basic Operations

Boolean and arithmetic operations with arbitrary sized RAM variables can be implemented in a bit serial fashion using the function generator. Thus we permit RAM variables to be combined using any of the standard boolean and arithmetic operations, provided their lengths are compatible.

Consider the following code sequence:

```
DSAstruct Sample {
   RAM A[3], B[2], C[2];
   } S;
S.A_i ← S.B_i + S.C_i;
```

The compiler might allocate the variables A, B and C in RAM as

A_i			B_i		C_i	
$R_i[0]$	$R_i[1]$	$R_i[2]$	$R_i[3]$	$R_i[4]$	$R_i[5]$	$R_i[6]$

Then the statement $A_i \leftarrow B_i + C_i$ would be expanded into code equivalent to

```
R_i[0] ← R_i[3] ⊕ R_i[5];
R_i[carry] ← R_i[3] ∧ R_i[5];
R_i[1] ← R_i[4] ⊕ R_i[6];
R_i[carry] ← (R_i[4] ∧ R_i[6]) ∨ (R_i[4] ∧ R_i[carry])
               ∨(R_i[6] ∧ R_i[carry]);
R_i[2] ← R_i[carry];
```

3.3 Selection and Querying

Groups of instructions are encapsulated in a loop-like block structure to indicate that they should be performed by a number of processing elements in parallel. An example of this form is:

```
ForEach i, ⟨boolean expression in i⟩ {
   ⟨forms involving i⟩
   }
```

The body forms are performed for each i that satisfies the boolean predicate. The simplest boolean predicate just indicates that i is an element of a particular set. For instance, the following code segment performs a calculation on the registers of the 32 even lines in the range 0 to 63.

```
ForEach i, i ∈ 0XXXXX0_2
   R_i[0] ← (R_i[0] ∧ ¬R_i[1]) ∨ R_i[2]
```

This particular code segment will be expanded into a **select** instruction to set the *SelectWord* to $\texttt{0XXXXX0}_2$ and a few operate instructions for the body of the loop,

```
select 0XXXXX0_2
R_i[temp] ← R_i[0] ∧ ¬R_i[1]
R_i[0] ← R_i[temp] ∨ R_i[2]
```

Odd sized DSA arrays, like the 100 entry **Table** given in Section 3.1, are dealt with by generating a DSA descriptor that is a union of selector bit strings. Then the body of the loop is repeated for each bit string. For instance, the sequence

```
ForEach $i$,
    $i \in 100XXXXXX_2 \cup 1010XXXXX_2 \cup 1110000XX_2$ {
  ⟨forms involving $i$⟩
  }
```

would be treated as:

```
ForEach $i$, $i \in 100XXXXXX_2$ {
    ⟨forms involving $i$⟩
    }
ForEach $i$, $i \in 1010XXXXX_2$ {
    ⟨forms involving $i$⟩
    }
ForEach $i$, $i \in 1110000XX_2$ {
    ⟨forms involving $i$⟩
    }
```

Notice that even though a **ForEach** loop evaluates its body at each line of the DSA array, the time required for the loop is typically $O(1)$, where the constant of proportionality is the time required to perform the body once. In the worst case, where the size of the DSA array is close to a power of 2, the loop will be performed $O(\log N)$ times for a DSA array with N lines.

Consider the following chunk of code.

```
ForEach $i$, $(i \in 0XXXXX0_2) \wedge (K_i \equiv \mathbf{Test})$ {
    $R_i[1] = R_i[1] \wedge R_i[2]$;
    printf ("Line %d matched", i);
    }
```

The predicate for this loop is a bit more complex. The body is performed for each of the even lines in the range 0 to 63 whose CAM's contents match **Test**. This predicate is expanded into three instructions: a **select** instruction that sets the *SelectWord* and a **match** instruction for $K_i \equiv$ **Test**. In addition the result of this match is stored in a register (R_i[**loop**]) for later use.

The first statement in the body is a simple **operate** cycle, except that it is only supposed to take effect on those lines that $R_i[\texttt{loop}] = 1$. This is accomplished by conditionalizing writes in the loop on the value of R_i[**loop**]. Thus the first line of the body expands into:

$$R_i[1] = (R_i[\texttt{loop}] \wedge R_i[1] \wedge R_i[2]) \vee (\overline{R_i[\texttt{loop}]} \wedge R_i[1]);$$

The final statement in the body actually expands into a loop. First an **operate** instruction is issued to store the contents of the R_i[**loop**] in the priority encoder register. Then a **readout** instruction occurs for each of the designated lines, followed by the code for the **printf** statement which uses the value returned by the successful **readout** instructions.

The set predicates available for use in a **ForEach** statement include multibit tests, multiple matches and arithmetic comparisons, which are discussed in Section 3.5.

3.4 State Transitions

A common use of the processing elements of the DSA is as a state machine. To make this a bit easier for the programmer and to make the resulting programs a bit more intelligible, we have decided to have the compiler allocate the binary patterns for states and work out the state transition equations. This is accomplished with two new types of statements.

A new set of states is introduced by the **States** form,

```
States $S \in \{up, down, sideways\}$;
```

This statement declares S to be a *state identifier* for each PE. The state of any particular PE is indicated by adding a subscript. Thus the state of the 5th processor is S_5. If we wanted to find all the processing elements which are in the *up* state, we would use the following code segment:

```
ForEach $i$, $S_i = up$
   printf ("Line %d is up, i);
```

Occasionally, computations may involve more than one set of orthogonal states. In this case, several state variables are declared, as in the following example.

```
States $S \in \{up, down, sideways\}$;
States $T \in \{red, yellow, blue\}$;
```

With these declarations, each PE could be in one of nine different states. We also say that the *S-state* of a processing element is S_i and that its *T-state* is T_i.

States can be changed by using the **NewState** form. The **NewState** form identifies which state variable is to be changed and has body consisting of a set clauses indicating how to change the state. For instance, we might have

```
States $S \in \{up, down, sideways\}$;
ForEach $i$, $i \in$ "XX...XX" {
   NewState $S$ {
      up:   $S_i \leftarrow down$;
      down:  if $K_i \equiv$ "X...X11X"
                 then $S_i \leftarrow up$;
                 else $S_i \leftarrow sideways$;
      otherwise:  $S_i \leftarrow sideways$;
      }
   }
```

If the compiler chose to use registers $R_i[0]$ and $R_i[1]$ to hold the state of each processing element, as follows:

State	$R_i[0]$	$R_i[1]$
up	0	0
down	0	1
sideways	1	X

There are three different binary inputs to the truth table for this state transition, the two state variables $R_i[0]$ and $R_i[0]$ and the result of the match $K_i \equiv$ "X...X11X", which we denote by M_i. This gives the following Karnough map:

	00	01	11	10
M_i	01	00	1X	1X
$\overline{M_i}$	01	1X	1X	1X

To minimize the the number of produce terms we use the following assignment of the X's.

	00	01	11	10
M_i	01	00	10	11
$\overline{M_i}$	01	10	10	11

Thus the original, state transition code is equivalent to

```
ForEach $i$, $i \in$ "XX...XX" {
    $R_i[0] \leftarrow ((K_i \not\equiv \text{"X...X11X"}) \wedge R_i[0]) \vee R_i[0]$;
    $R_i[1] \leftarrow \overline{R_i[1]}$;
    }
```

which, though somewhat shorter textually, is significantly less clear than the state transition code given earlier.

3.5 Arithmetic Comparisons

One of the fundamental applications of the data structure accelerator is searching tables quickly. The content addressable memory accelerates searches involving boolean patterns, while the function generator accelerates searches involving arithmetic patterns. For example, we can determine those lines of a DSA that contain a number lying between given bounds, or the line of the DSA that contains the largest quantity in time independent of the number of lines being searched.

For simplicity we assume that each line of the DSA contains precisely one m-bit key. This key can lie either in the CAM portion of the DSA or in the RAM portion. In this section we assume the key lies in the CAM, but trivial modifications of the algorithms enable it work with the key in the RAM. We consider two fundamental operations, comparison with a given quantity and finding the largest element of a set of keys.

The following function identifies each line whose key is greater than **LowerBound**. This is done by examining each of the bits of the key in sequence. Each DSA line can be one of three states: *greater*, *lesser* and *unknown*. In state *lesser* the entry is known to be less than the key; in the *greater* state it is known to be greater than the key, and in state *unknown* we still don't know. If an entry is in state *unknown* then its leading bits match the leading bits of the key that have been presented so far.

All words are initially placed in the *unknown* state. We compare the contents of the CAM with **LowerBound** one bit at a time, changing the state of the line as necessary. The state transition diagram is shown in Figure 2. At the end of m cycles, any word still in state *unknown* is equal to the key.

The following program uses two special arrays. **BitMask**[n] contains a 1 in the n^{th} bit position, from highest to lowest. Thus **LowerBound** $\wedge$ **BitMask**[n] selects the n^{th} bit from **LowerBound**. **FieldMask** is similar, but has the n highest bits set.

```
Compare(Array, LowerBound) {
    States Array.$S \in \{greater, lesser, unknown\}$;
    ForEach $i$, $i \in$ Array {
        Array.$S_i \leftarrow unknown$;
        for $0 \leq n <$ MatchWidth {
          NewState $S$ {
            unknown:
              if Array.$K_i \equiv$
                    (LowerBound $\wedge$ FieldMask[$n$])
                then Array.$S_i \leftarrow unknown$;
              else if $0 =$ LowerBound $\wedge$ BitMask[$n$]
                    then Array.$S_i \leftarrow greater$;
              else Array.$S_i \leftarrow lesser$;
            otherwise:  Array.$S_i \leftarrow$ Array.$S_i$;
            }
          }
        }
    }
```

This and similar routines can be used by the compiler to implement arithmetic predicates in **ForEach** statements. For instance, one might want to use the DSA to represent a large set of one dimension intervals. The following code segment would then be used to find those intervals that contain the origin.

```
DSAstruct Sample {
    CAM $L[16]$, $R[16]$;
    } S;

ForEach $i$, (S.$L_i < 0$) $\wedge$ (S.$R_i > 0$)
    printf("Interval %d contains the origin.",
        i);
```

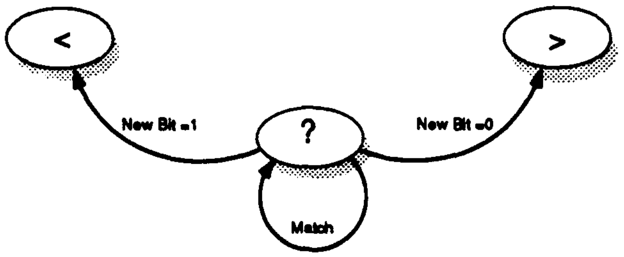


Figure 2: State Diagram for Comparison

For example, we might want to use the DSA to represent a set of intervals from 0 to $2^{16}-1$. Each interval would be represented as one line of a $(32, n)$ DSA, with half of K_i used to hold the lower li :it of the interval and half for the upper limit. A modest sized data structure accelerator could then contain a rather large set of these intervals. Using this algorithm, we can determine the intervals in this set that intersect a given interval in time linear in the size of the intersection. This technique trivially extends to two or more dimensions.

3.6 A Space/Time Trade^ff

One unfortunate effect of the linear nearest neighbor interconnection network of the data structure accelerator is that the (graph theoretic) diameter of a data structure accelerator of n lines is n. Thus computations that require data in distant lines be combined can be quite slow. The boolean n-cube type networks used by the Connection Machine [3] have diameter $\log n$ and thus can perform somewhat better with these algorithms. Unfortunately, boolean n-cube networks do not scale to large numbers of processing elements. In fact, networks with diameter less than $n^{1/3}$ cannot be embedded in 3-space in a uniformly scalable fashion. Power distribution and heat dissipation considerations raise this bound to $n^{1/2}$ and packaging considerations to n. Thus algorithms that require a high degree of communication among $O(n)$ processing elements will require more than $O(n)$ "real estate" in realizable systems.

This section shows how to solve such a problem using $O(n^2)$ lines of a DSA. Notice that while fewer processors could be used if a smaller diameter network were used, it is not clear that less total hardware would be required.

Consider the following problem: Given a set of n integers $\{a_0, \ldots, a_{n-1}\}$, find those pairs that have the minimum difference. The brute force approach of comparing all pairs of integers requires $O(n^2)$ operations. However, this can be reduced to $O(n \log n)$ operations by first sorting the integers an then comparing their neighbors. Using a DSA we can solve this problem using $O(n)$ space and $O(n)$ time in the following fashion. First, store each a_i in a line of the DSA. Then, in parallel, compute the difference between the contents of each line and a_0. Repeat this for each a_j retaining the smallest difference. This will take $O(n)$ operations. Finally, the smallest difference is determined using the techniques of Section 3.5. This requires $O(1)$ operations. Thus this problem can be solved in $O(n)$ time by using $O(n)$ lines of a DSA. The computational complexity of this solution is still $O(n^2)$ ($O(n)$ processors and $O(n)$ time), the same as the simple algorithm, but we have achieved a modest speed up ($O(\log n)$) while continuing to use a straightforward algorithm.

An alternative approach is to store in each of n^2 lines of the DSA the pairs (a_i, a_j). Then the n^2 differences can be computed in parallel using $O(1)$ operations. This may be useful approach if many such calculations with the same set of a_i are to be performed. The only problem is to get the data into the DSA efficiently.

Observe that the n^2 entries in the DSA can be written using only $O(n)$ operations by using the selector carefully. Assume that $n = 2^k$, We begin by writting a_i in the n lines beginning with line in. Then write a_i in lines i, $n+i$, $2n+i$ and so on. Each of these two passes requires $O(n)$ write operations so the entire n^2 array can be set up in $O(n)$ time.

The following code fragment implements this procedure for a 4 x 4 array. For simplicity, we have (unrealistically) assumed that each of the a_i is a single bit. Notice that each line is a single DSA instruction. Figure 3 illustrates the operation of this technique when applied to 2 x 2 case.

```
ForEach i, i ∈ "00XX"
    R_i[left] ← a_0;
ForEach i, i ∈ "01XX"
    R_i[left] ← a_1;
```

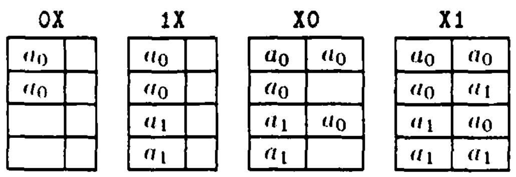


Figure 3: Intermediate states while creating a cross product

```
ForEach $i$, $i \in$ "10XX"
    $R_i$[left] ← $a_2$;
ForEach $i$, $i \in$ "11XX"
    $R_i$[left] ← $a_3$;
ForEach $i$, $i \in$ "XX00"
    $R_i$[right] ← $a_0$;
ForEach $i$, $i \in$ "XX01"
    $R_i$[right] ← $a_1$;
ForEach $i$, $i \in$ "XX10"
    $R_i$[right] ← $a_2$;
ForEach $i$, $i \in$ "XX11"
    $R_i$[right] ← $a_3$;
```

4 Problems in Computational Geometry

In this section we demonstrate how the data structure accelerator can be used to solve several problems in computational geometry. Since problems in computational geometry typically arise as a component of other applications (such as VLSI or mechanical CAD), we think the use of the data structure accelerator is particular appropriate. It is relatively inefficient to design specialized hardware to solve the computational geometry problems that arise in CAD, because they are just one component of a larger computational problem. And yet there is a huge amount of potential parallelism to be exploited. The data structure accelerator provides that parallelism in a fashion that is not specialized to the problems of computational geometry. At the same time, these techniques require using the data structure accelerator in tandem with regular processing elements. This is precisely the type of cooperative heterogeneous computation discussed in the introduction.

4.1 Convex Polygon Inclusion

The convex polygon inclusion problem is relatively straightforward. A convex polygon is described by the sequence of its vertices, as shown in Figure 4(a). We are to determine if a given trial point is contained within the polygon. The basic relationship we use is illustrated in Figure 4(b). If we denote the x and y coordinates of the point P_i by x_i and y_i, the signed area of the triangle $P_1P_2P_3$ is

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2+x_2y_3+x_3y_1-x_1y_3-x_2y_1-x_3y_2.$$

The sign of the area is positive if the points P_1, P_2 and P_3 are arranged counterclockwise in the plane [5]. Thus in Figure 4(b),the triangle $P_1P_2P_3$ has positive area, while the triangle $P_1P_2P_4$ has negative area.

To determine if a trial point P is contained within a polygon we check that the triangles formed by P and each edge of the polygon have positive area. This is easily done by assigning each edge of the polygon to a line of the DSA:

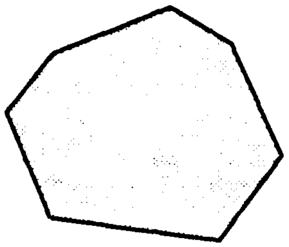
```
DSAarray LineSegment {
    RAM Lx[$\ell$], Ly[$\ell$], Rx[$\ell$], Ry[$\ell$], ;
        Area[2$\ell$];
    };
```

We have allocated four ℓ-bit quantities to hold the coordinates of the endpoints, and one 2ℓ-bit quantity for the area of the triangle in the computation.

```
PolygonInclusion(Edges, Px, Py)
DSAarray LineSegment Edges[]; {
  ForEach $i$, $i \in$ Edges {
    Edges.Area$_i$
      ← Edges.Lx$_i$ $\times$ Edges.Ry$_i$ + Edges.Rx$_i$ $\times$ Py
        + Px $\times$ Edges.Ly$_i$ − Edges.Lx$_i$ $\times$ Py$_i$
        − Edges.Rx$_i$ $\times$ Edges.Ly$_i$ − Px$_i$ $\times$ Edges.Ry$_i$;
    if $\exists j$.(Edges.Area$_j$ < 0)
      then return( "Outside");
      else return( "Inside");
    }
  }
```

The time required by this algorithm is independent of the number of edges of the polygon(s), but due to the multiplications in the area computation, quadratic in the number of bits required to represent the coordinates of the vertices $O(\ell^2)$. We could say that the time is $O(\ell \log \ell)$ by using an FFT algorithm, but this would only be of theoretical interest and ignores the performance cost of getting a larger algorithm to the DSA. In addition, we must count the preprocessing time required to load the n vertices into the DSA, which is $O(\ell n)$, since each point has size $O(\ell)$. The time to check m trial points grows to $O(m\ell^2)$), while the preprocessing time remains fixed (using classical multiplication).

If the number of trial points is large, or the same trial points are to be used repeatedly with different sets of polygons, we can store the trial points in the DSA and



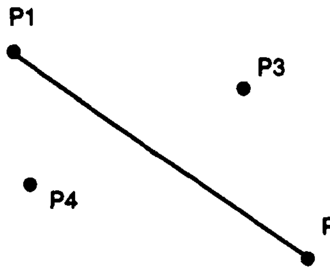


Figure 4: Polygon Inclusion

perform the computation for different polygons. In this case the preprocessing time becomes $O(\ell m)$ while the computing time becomes $O(n\ell^2)$).

Classical algorithms [5] require $O(\ell n)$ time for preprocessing and answer the inclusion question for m trial points in time $O(\ell^2 m \log n)$. For comparison, these results are summarized below.

	Preprocessing	Query
vertices in DSA	ℓn	$m\ell^2$
trial points in DSA	ℓm	$n\ell^2$
classical	ℓn	$\ell^2 m \log n$

The DSA approach uses a straightforward algorithm and achieves somewhat better performance than the classical techniques. The following section discusses a variant of this problem where the test points lie in a regular grid.

4.2 Polygon Filling

A common primitive for a number of geometric algorithms is *polygon filling*. In the two dimensional case, we are given a rectangular section of the plane discretized into an $m \times n$ grid. Within this grid are marked the boundaries of a number of connected regions. The polygon filling problem is to identify the regions in which each point of the grid is contained.

A simple example of this problem arises in computer graphics where the regions might represent homogeneous regions of an image, *e.g.* surfaces of objects. When the image is presented on the screen, each pixel within each region needs to be painted with the same color.

Figure 5 shows a two dimensional region embedded in a grid. Each dot represents a single processing element of a two dimensional DSA. The diagonal lines form the true boundary of the region, while the black dots indicate the boundary on the grid. Notice that each black dot lies on or within the boundary of the region. Given such a boundary we can propagate a seed node outward until it reaches a boundary. This is illustrated in Figure 5 where the seed node is at $(6,5)$. At t_0 it is the only node marked. Between t_i and t_{i+1} each marked node propagates a mark to each of its four neighbors if they are (1) not already marked and (2) not a element of the boundary. The time at which each node is marked is given in the figure. In this case every node in the region that is orthogonally connected to $(6,5)$ is marked in 7 units of time.

The data structure used to model the grid is defined as follows.

```
DSAstruct FillGrid {
   States $S \in \{interior, exterior, boundary, unknown\}$;
   } Grid[$n$, $n$];
```

Each node in the grid can be in one of four states: *interior*, *exterior*, boundary and *unknown*. Initially each node is placed in the *unknown* state. The boundary is then defined by place each node on or just inside the boundary to the *boundary* state. The orientation of the boundary is defined by setting one node inside the region to the *interior* state. This node serves as a seed that spreads throughout the region.

The following block of code then propagates the seed throughout the interior.

```
Propogate (Grid) {
   for $\ell < n$ {
     ForEach $(i,j)$, $(i,j) \in$ Grid {
        NewState Grid.$S$ {
          unknown:
             if (Grid.$S_{i+1,j}$ = interior)
                 $\vee$(Grid.$S_{i,j+1}$ = interior)
                 $\vee$(Grid.$S_{i-1,j}$ = interior)
                 $\vee$(Grid.$S_{i,j-1}$ = interior)
```

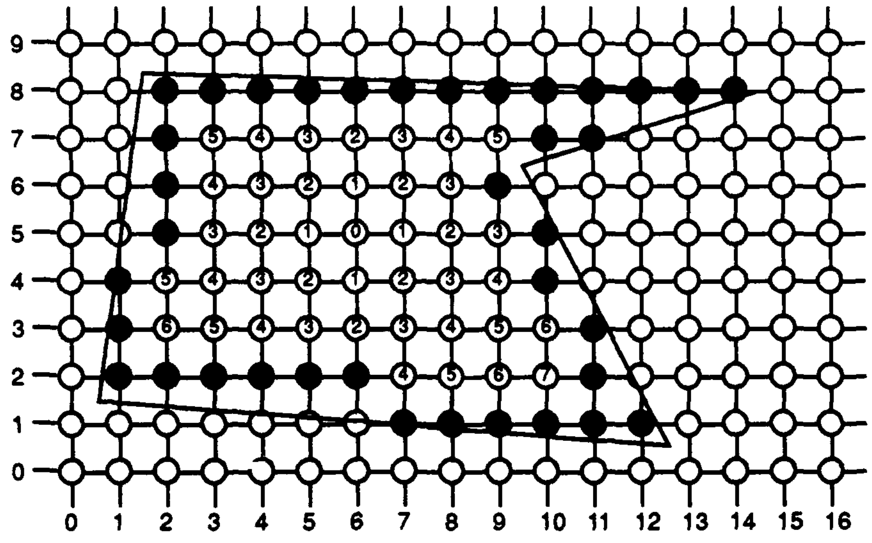


Figure 5: Sample Image

```
                then Grid.$S_{i,j}$ ← interior;
              else Grid.$S_{i,j}$ ← unknown;
            otherwise:  Grid.$S_{i,j}$ ← Grid.$S_{i,j}$;
          }
        }
      }
    }
```

This routine assumes the orthogonal distance between any two connected nodes is no more than n. This is the case for convex polygons, but serpentine (concave) polygons can be created whose interiors have minimal orthogonal paths of length $O(n^2)$.

5 Conclusions

This paper has discussed a SIMD architecture that is designed to be used as an integral component of a heterogeneous highly parallel machine. We have described a few basic algorithms for using the data structure accelerator and provided a language for describing other algorithms. The major observation we make is not that the DSA yields enormous speed-ups over optimal sequential algorithms, but rather we get a modest speedup over the exceedingly complex, optimal algorithms by using a straightforward algorithms on the DSA. Thus we claim to have improved the complexity-speed/cost product.

This paper benefited from the comments of Laurie Hendren, James Stewart and Steve Vavasis. Paul Chew has corrected a number oversights and generally improved content and presentation of this paper.

References

[1] Sharon Marie Britton. 8k-trit Database Accelerator with error detection. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, February 1990.

[2] W. Daniel Hillis and Guy Lewis Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[3] W. Danny Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.

[4] Jerry L. Potter. *The Massively Parallel Processor*. MIT Press Series in Scientific Computation. MIT Press, Cambridge, MA, 1985.

[5] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Texts and monographs in computer science. Springer-Verlag, New York, 1985.

[6] Jon P. Wade, Peter Osler, Richard E. Zippel, and Charles Sodini. The MIT Database Accelerator: 2k-trit circuit design. In *1987 Symposium on VLSI Circuits*, Karuizawa, Japan, January 1987.

[7] Jon P. Wade and Charles G. Sodini. Dynamic cross-coupled bitline content addressable memory cell for high density arrays. *IEEE Journal of Solid State Circuits*, SC-22(2):119-121, February 1987.

[8] Jon P. Wade and Charles G. Sodini. A ternary content addressable search engine. *IEEE Journal of Solid State Circuits*, SC-24(4):1003-1013, August 1989.

[9] C. Weems, S. Levitan, and Caxton Foster. Titanic: A VLSI based content addressable parallel array processor. In *Proceedings of 1982 International Conference on Custom Circuits*, pages 236–239. IEEE, 1982.